ROBiT - A Binary Optimization Anti-Plagiarism Method

Roberta Robert[1], **Bruno Castro da Silva**[2], **Jeferson Campos Nobre**[1],

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

[2]College of Information and Computer Sciences – University of Massachusetts
Amherst, USA

{rrobert, jcnobre}@inf.ufrgs.br, bsilva@cs.umass.edu

Abstract. *The widespread availability of Large Language Models (LLMs) has significantly lowered the barrier to committing code plagiarism. However, most existing anti-plagiarism tools remain vulnerable to modern evasion strategies, including syntactic transformations and generative code rewriting. Prior work shows that such transformations can effectively bypass clone detectors that rely on syntactic or semantic representations. While binary optimization is a known technique in malware obfuscation, its potential for plagiarism detection has been largely overlooked. We introduce a hybrid detection method that combines source-level syntactic analysis with binary-level comparison, leveraging both standard compilation outputs and binaries generated with optimization flags. These optimizations act as a reverse filter, eliminating syntactic manipulations added to code artifacts and revealing structural similarities with the original binary. Our empirical evaluation confirms that optimized binaries exhibit patterns that correlate strongly with their original source code. The proposed method demonstrates high effectiveness in detecting plagiarism, even when the source code has undergone aggressive syntactic transformations. This technique serves as a robust and complementary extension to existing syntactic anti-plagiarism systems, offering deeper insight into semantic and structural code similarity.*

1. Introduction

The concern over potential plagiarism in computer science classes is not new. One early article discussing the implications of plagiarism and presenting one of the first similarity analysis techniques dates back to 1986 [Faidhi and Robinson 1987]. The shift to remote learning formats has affected both how instructors evaluate assignments and how students approach them. One notable change in practical courses is how code reviews are conducted: due to the increase in class sizes, the average time available for manual review of programming assignments has decreased [Duracik et al. 2020], leading to broader adoption of automated anti-plagiarism tools [Aiken 2004].

The rapid rise of tools based on Large Language Models (LLMs), such as ChatGPT, has also reshaped the discussion around code creation. Recent debates focus on LLM-generated code being considered innate [Geng et al. 2023], meaning that a student with no prior knowledge of the problem can produce a seemingly original solution using prompt engineering.

These two factors have brought us to a moment where plagiarism automation techniques are widely available and easy to use, often requiring minimal technical expertise.

Even in contexts where manual code inspection complements automated tools, detecting plagiarism remains difficult. Mossad [Devore-McDonald and Berger 2020] exemplifies this challenge by generating multiple code variants that preserve readability and semantics.

Despite the widespread use of plagiarism detection systems, particularly `MOSS` [Aiken 2004], these tools generally rely on syntactic analysis algorithms [Schleimer et al. 2003, Liu et al. 2006, Wise 1996, Prechelt et al. 2002, Nichols et al. 2019]. A broad array of evasion strategies has emerged to target syntactic-based detection [Devore-McDonald and Berger 2020, Zhang et al. 2021, Biderman and Raff 2022]. Mitigation efforts often focus on pedagogical or policy-based solutions rather than improving the underlying detection algorithms themselves.

This work presents a new hybrid plagiarism detection method called ROBiT (Reverse Optimization Binary Transformation), based on leveraging compiler optimization techniques during the build process. We demonstrate that this approach can identify plagiarism cases involving complex syntactic transformations to the original code — cases that typically evade most existing analyzers. Our experiments confirm a strong correlation between optimized binaries and their original source code binary, regardless of how the problem was implemented at the code level. The method proposed here is intended to complement existing anti-plagiarism tools.

This paper is organized as follows: Section 2 presents the technical background and related work. Section 3 describes the proposed method (ROBiT). Section 4 details the experimental setup and results. Section 5 compares our approach with related methods. Finally, Section 6 presents our conclusions and future work.

2. Background and Motivation

This section discusses *(i)* how syntactic and semantic characteristics interact within the scope of source code and its binaries; *(ii)* how the process of optimizing binary code—which underpins our method—works; and *(iii)* how our chosen binary comparison method operates. Our work focuses specifically on processes related to the construction and compilation of C source code. Thus, the topics discussed below pertain to static code analysis involving pre-runtime compilation. We do not explore execution-time compilation or interpreted languages at this work.

2.1. Syntactic and Semantic Analysis Algorithms

Evaluating source code to identify potential plagiarism is a complex task, as its structures can be categorized from lexical, syntactic, and semantic perspectives. Due to this complexity, auxiliary methods are necessary to translate these characteristics from a C source code into alternative formats, enabling more effective comparisons. These methods, known as *intermediate representation algorithms*, generate various representations of a source code at different abstraction levels throughout the compilation process. Although multiple types exist, we focus on four widely used in plagiarism studies: tokenization, abstract syntax trees (ASTs), control flow graphs (CFGs), and program dependency graphs (PDGs). Each of these transforms the high-level source code (e.g., in C) into representations that highlight different aspects of program structure. Such representations are essential to the compilation process and support plagiarism detection.

Many source code plagiarism detectors rely primarily on tokenization and AST analysis. Tokenizing a program allows for basic lexical comparisons, while ASTs enable a deeper structural syntactic analysis, even identifying plagiarism despite superficial changes. Tools like MOSS [Schleimer et al. 2003] and JPlag [Prechelt et al. 2002] use these techniques to identify significant similarities between programs.

2.2. Anti-Plagiarism Tool - MOSS

MOSS (Measure of Software Similarity [Schleimer et al. 2003]) is one of the most widely used plagiarism detection tools, especially in academic contexts. It applies a fingerprinting-based algorithm to detect similarities between source codes by generating fingerprints based on sequences of meaningful tokens, while ignoring trivial elements.

These fingerprints are compared across documents after a normalization process that removes comments and whitespace. This approach allows MOSS to efficiently identify matching segments across different files, helping to detect potential plagiarism. For the work presented here, we will be using it as a baseline for comparison, as the technique used to show the plagiarized code on MOSS attacks a different intermediate representation layer than the one we're using in our method.

2.3. Syntactic and Semantic Transformations

To assess code similarity and identify plagiarism, we highlight *Code Clone Detection* techniques. These approaches examine semantic and syntactic transformations to detect copied sections, even when code has been manipulated - one example is cases where there is a dispute about intellectual property protection. As our research intersects with this field, we adopt selected techniques to illustrate common evasion strategies—specifically, transformations designed to circumvent code-cloning analysis that can be applied into plagiarized code as well.

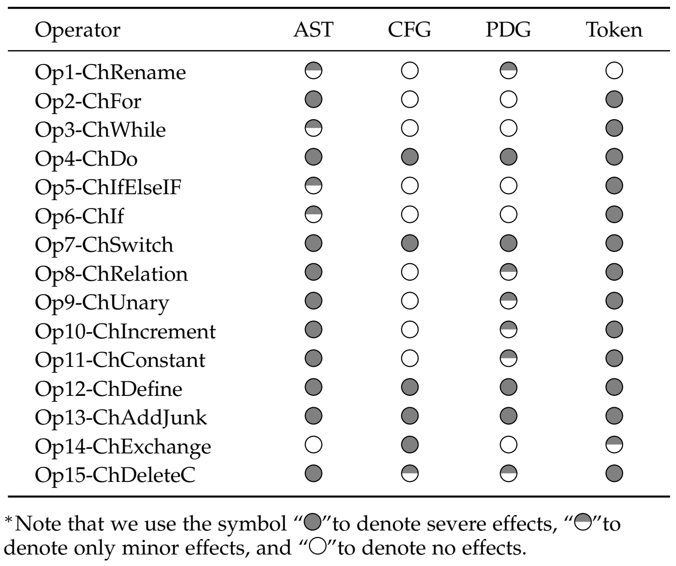
Zhang et al. proposed a similarity grading scheme based on four levels:

- **Type I**: Identical code except for differences in comments, indentation, and layout.
- **Type II**: Type I plus changes to variable names, data types, or function names.
- **Type III**: Structural changes such as reordering or modifying declarations.
- **Type IV**: Semantic-level similarity, where code is altered syntactically but maintains the same behavior.

They also proposed 15 atomic transformations that can evade various levels of syntactic and semantic similarity detection. We selected four of these transformations to generate our test cases due to their semantic evasiveness:

- **Op4-ChDo**: Converts *do-while* loops to *while* loops.
- **Op7-ChSwitch**: Replaces *switch* statements with equivalent *if-else* chains.
- **Op12-ChDefine**: Alters variable declarations and initializations.
- **Op13-ChAddJunk**: Adds dead code that doesn't affect execution.

Applying these transformations alters both syntactic features (e.g., code length and structure) and intermediate representations such as CFGs and PDGs. As shown by Zhang et al. at Figure 1, these modifications often evade both syntactic and semantic detection tools by disrupting underlying representation graphs and control flow—which is exactly the type of obfuscation our method aims to address.



Operator	AST	CFG	PDG	Token
Op1-ChRename	◒	○	◒	○
Op2-ChFor	●	○	○	●
Op3-ChWhile	◒	○	○	●
Op4-ChDo	●	●	●	●
Op5-ChIfElseIF	◒	○	○	●
Op6-ChIf	◒	○	○	●
Op7-ChSwitch	●	●	●	●
Op8-ChRelation	●	○	◒	●
Op9-ChUnary	●	○	◒	●
Op10-ChIncrement	●	○	◒	●
Op11-ChConstant	●	○	◒	●
Op12-ChDefine	●	●	●	●
Op13-ChAddJunk	●	●	●	●
Op14-ChExchange	○	●	○	◒
Op15-ChDeleteC	●	◒	◒	●

*Note that we use the symbol "●"to denote severe effects, "◒"to denote only minor effects, and "○"to denote no effects.

Figura 1. Transformation operators and their impact on the four frequently used code transformations [Zhang et al. 2021].

2.4. Binary Optimization Techniques

In malware research, *binary obfuscation* refers to techniques that obscure binaries to evade detection tools, including plagiarism checkers [Poornima and Mahalakshmi 2023]. One such technique, binary optimization, uses compiler transformations to modify binary signatures without changing program behavior.

These optimizations are applied via compiler flags, which trigger a set of transformations during compilation. While some obfuscation techniques aim to increase divergence from source code, our method leverages these optimizations to achieve the opposite effect: to reverse the distortions introduced by code plagiarism.

2.5. GCC — Optimization Flags

The GCC compiler [Free Software Foundation 2024] provides several levels of optimization: *-O0*, *-O1*, *-O2*, and *-O3*. Each successive level includes the flags from the previous ones and introduces more aggressive transformations. *-O0* performs no optimization and reduces compilation time. We selected *-O3* as the basis for our method based on findings from Ren et al. [Ren et al. 2021], who demonstrated the substantial impact of this level compared to *-O0*. *-O3* performs broad, deep transformations that reduce structural noise and improve binary consistency.

Some transformations, such as **ChAddJunk** and **ChDefine**, tend to produce binaries that are close to the original non-optimized binary, making them difficult to distinguish using only binary comparison. Therefore, our experiments do not rely solely on comparing non-optimized and optimized versions of the same file. Instead, we also incorporate comparisons between the original and transformed source codes—both compiled with and without optimization—to assess how compiler optimizations affect similarity across different code variants. Our goal is to determine whether these optimizations can act as a "reverse filter,"neutralizing superficial obfuscations and restoring structural similarity in the resulting binaries. By comparing both the unoptimized and optimized binaries of the original and altered source codes, we can better understand the role of optimization in exposing semantically equivalent but syntactically obfuscated code.

2.6. Binary Comparison - Edit Distance

To analyze and compare binaries, we rely on similarity detection techniques like edit distance. These methods, such as Levenshtein distance [Levenshtein 1965], calculate the minimal number of atomic operations (insertions, deletions, substitutions) needed to transform one string (or sequences of bytes in a binary code) into another. The smaller the edit distance, the greater the similarity.

Binary comparison applications (*binary diffing*) frequently use variations of this method. In our case, we chose Radiff2 [Radare2 Team 2024], which offers byte-level similarity comparison through various functions. We employ the *-ss* option exclusively, as our preliminary testing showed that higher-level comparisons (e.g., at the assembly level) were either already well-explored or fell outside the scope of our approach. The results from Radiff2 are used to assess binary similarity and guide our plagiarism detection analysis.

3. `ROBiT`: An Anti-Plagiarism Analysis Method

`ROBiT` is the anti-plagiarism method proposed in this work. It aims to identify cases of plagiarism by analyzing sets of C programs and measuring the similarity between their respective binary codes, generated through GCC compilation with optimization flags.

Given a set of source codes submitted for analysis, our method first compiles the C programs using the *-O3* optimization flag. It then calculates the Levenshtein distance between all binary file pairs using the Radiff2 technique. Once all program pairs have been compared, the results are stored in a text file for later consultation and analysis.

For each pair of binaries, the method generates two values: *(i)* a similarity score, expressed as a percentage inversely proportional to the Levenshtein distance (i.e., the smaller the distance, the higher the similarity); and *(ii)* the Levenshtein distance itself. These metrics are correlated such that a greater edit distance implies a lower likelihood of plagiarism.

It is important to note that using GCC *-O3* optimizations effectively results in a type of "semantic normalization" in both binary codes. If two C programs implement the same functionality but use different commands and instructions, the optimization process tends to map the corresponding source codes to similar binaries. Intuitively, the low-level representations, induced by the advanced optimization process, tend to be similar in that the optimization preserves the *functionality* implemented by each C code, regardless of *how* such functionality was implemented in the high-level language. This possibility allows comparisons of the resulting binaries to identify plagiarism cases more consistently on average, as it can identify functional similarities even when the corresponding source codes have undergone complex syntactic changes.

All the experiments and the related codebase of this method can be found at the project's github page [1].

[1]https://github.com/dujour/ROBiT

3.1. Experimental Methodology

The experiments proposed in this section evaluate the performance of our method compared to two alternatives: a variant created as a baseline called `ROBiT-Simple` (which uses not-optimized binaries for comparison), and `MOSS` (one of the most widely used anti-plagiarism tools).

The source code language used for the experiments is the C language, because their broad adoption into introductory Computer Science courses, as well as the easier capability to debug and see the CFG's behavior in low level translations. No additional source code language will be used into these experiments.

3.1.1. Choice of Base Problems

For the experiments, we selected two typical programming problems from introductory courses:

- **Program 1**: *"Create a program that reads the initial mass of a material (in grams) and calculates how long it takes for this mass to reduce to less than 0.5 grams, assuming that the mass is reduced by half every 50 seconds. The program must verify if the initial mass is valid (greater than or equal to 0.5 grams) and print the final mass, along with the total elapsed time in hours, minutes, and seconds."*
- **Program 2**: *"Write a program in C that calculates the approximate value in radians and degrees of arctan(x), where the user enters the value of x. The program must verify if the value of x is within the valid range (in this case, -1 ¡ x ¡ 1). If it is within the valid range, the user must enter the number of terms for the sum. The program then must calculate the arctan(x) using the Taylor series up to the specified number of terms and print the result in radians and degrees."*

Typically, these exercises have a simple, well-defined scope, which implies low variability in how they can be solved—especially in terms of issues such as variable name choices, definition of limits, etc. These limitations are relevant as they permit the segregation of the results of syntactic and semantic transformations applied to a base source code to produce plagiarism, facilitating the investigation of their impact on the results.

3.1.2. Generation of Codes Used in Experimental Analysis

To generate the source codes used in our experiments, we employed ChatGPT ([Yenduri et al. 2023]). ChatGPT can provide unique and different responses to each user's question/prompt.

In our experiments, we predetermined which characteristics the corresponding source code should have using prompt engineering. For example, we would define that it should create code versions using *switch-cases* and *do-whiles*, since those are two subsequent syntactic transformations we wanted the code variations to have, either isolatedly or applied together. Having this base code created, it was then created 4 different versions of the original code, each one with one of the transformations. We also created one version with all the transformations applied simultaneously to introduce more variability in our comparisons. The 6 source codes for each experiment are also evaluated through MOSS to obtain a syntactic similarity baseline, so we can use it as a ground of comparison of eficacy.

3.1.3. Creating **ROBiT-Simple**

To assess the role of optimization, we developed a simplified variant of our method called ROBiT-Simple. It performs the same binary comparisons as ROBiT but compiles all source codes using the default optimization level (*-O0*). The resulting binaries are compared using Radiff2, and the output is stored in the same structured format, allowing direct comparison between optimized and unoptimized analysis results.

4. Experimental Results

This section presents a qualitative and quantitative analysis of the empirical results obtained to validate the effectiveness of our method against two alternative techniques: (i) the ROBiT-Simple method, which also relies on binary comparison via *edit distance* but does not incorporate optimization during compilation; and (ii) MOSS, one of the most widely used syntactic-based anti-plagiarism analyzers in academic contexts.

The following results are grouped into three categories, each accompanied by discussions, interpretations, and observations. Specifically, we present:

1. A comparison between ROBiT and MOSS (*Section 4.1*);
2. A comparison between ROBiT and ROBiT-Simple, which omits compiler optimizations (*Section 4.2*);
3. An analysis of method performance in challenging scenarios involving syntactic and semantic transformations applied to original source codes (*Section 4.3*).

4.1. **ROBiT** vs. **MOSS**

We begin by comparing ROBiT with the MOSS algorithm. We measured the similarity percentage reported by each tool when analyzing five transformed versions (plagiarized variants) derived from two base C programs. Higher similarity percentages indicate stronger detection capability.

The results for each of the two base programs are referred to as *Experiment 1* and *Experiment 2*, presented in Figures 2a and 2b, respectively. In all cases, the similarity values shown correspond to the average reported similarity over 15 pairwise comparisons, which include base vs. modified versions and cross-comparisons between transformed codes.

In both experiments, ROBiT consistently outperformed MOSS. The difference in similarity detection rates is notable, especially in cases where MOSS either failed to detect plagiarism or returned very low similarity scores. When MOSS does report a potential match, the confidence is typically low, suggesting high susceptibility to syntactic and structural code obfuscation techniques. In contrast, ROBiT retains high detection rates even under transformation.

It is important to reiterate that the techniques used to modify base source codes and generate plagiarized files, in both experiments, correspond to techniques that,

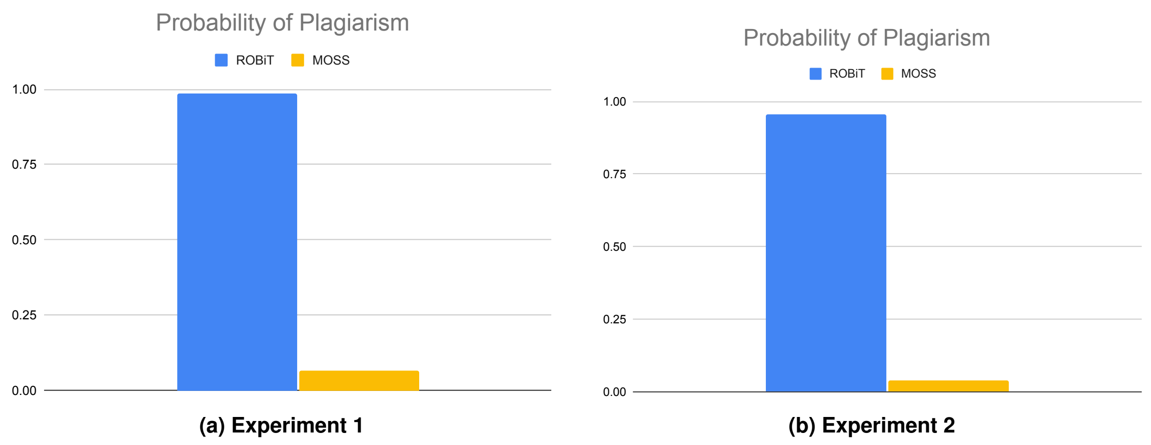


Figura 2. Comparison of plagiarism probabilities identified by the techniques `ROBiT` and `MOSS` across two experiments.

although relatively simple (e.g., replacing one loop structure with another), significantly affect various existing anti-plagiarism methods. *For this reason, the improvements observed in the level of detection of plagiarism identified by our method, when compared with the `MOSS` algorithm (which is widely used in many academic environments), reinforce the importance of our contribution to the field.*

4.2. `ROBiT` vs. `ROBiT-Simple`

The second empirical result discussed in this section corresponds to the comparison of binary pairs (in both experiments) for the case of the algorithms `ROBiT` and `ROBiT-Simple`. Both techniques are based on binary code comparison: `ROBiT` (our method) operates on binaries generated through optimization compilation processes, while `ROBiT-Simple` analyzes only non-optimized binaries.

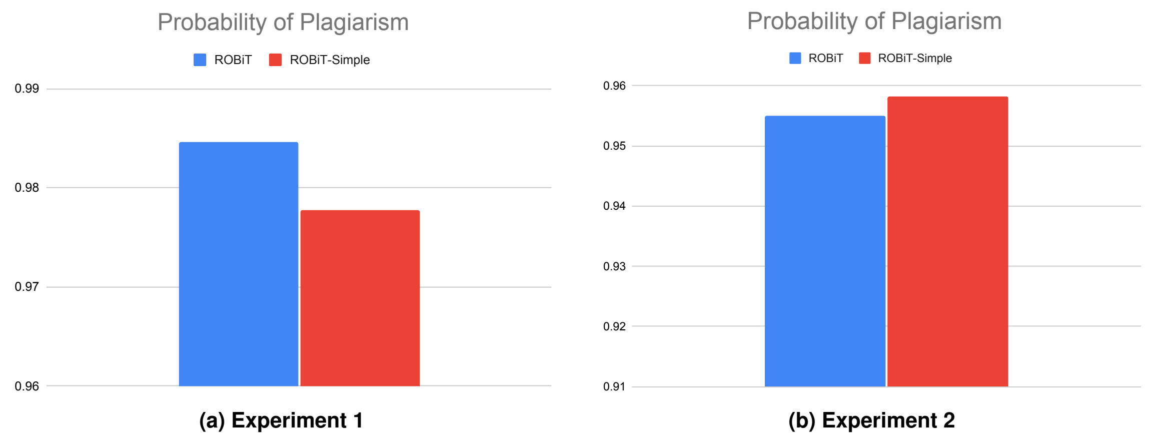


Figura 3. Comparison of plagiarism probabilities identified by the techniques `ROBiT` and `ROBiT-Simple` across two experiments.

Figures 3a and 3b show that in Experiment 1, our method—`ROBiT` —achieved a slight advantage over `ROBiT-Simple`. In contrast, `ROBiT-Simple` demonstrated marginally better performance in Experiment 2. This variation is likely influenced by the structural characteristics of the base programs used in each experiment. For example, one base program did not include *switch-case* clauses, making certain transformations—such as **Op7-ChSwitch**—inapplicable. As a result, each base program is affected differently by the available set of syntactic and semantic transformations. We hypothesize that these structural differences contribute to the relative performance of each detection technique, although the observed variations are not statistically significant.

To expand the scope of Experiment 1, which already includes a plagiarism file generated through the application of *all* possible syntactic and semantic transformations, we added a second plagiarism file with this same characteristic. These programs are called, respectively, *p1-all* and *p1-all-2*. Although these programs include the same types of syntactic constructions and C language commands, their source codes naturally differ slightly. However, when analyzed by `ROBiT`, due to the use of optimizations, the resulting binary codes tend to be *similar*. On the other hand, these small differences in their source codes can generate substantially different binary files if compiled without optimizations. In this case, it is natural that the `ROBiT` method performs slightly better than `ROBiT-Simple` in terms of its ability to detect similarities. This difference in performance between the two methods can be observed in the first row of Figure 4.

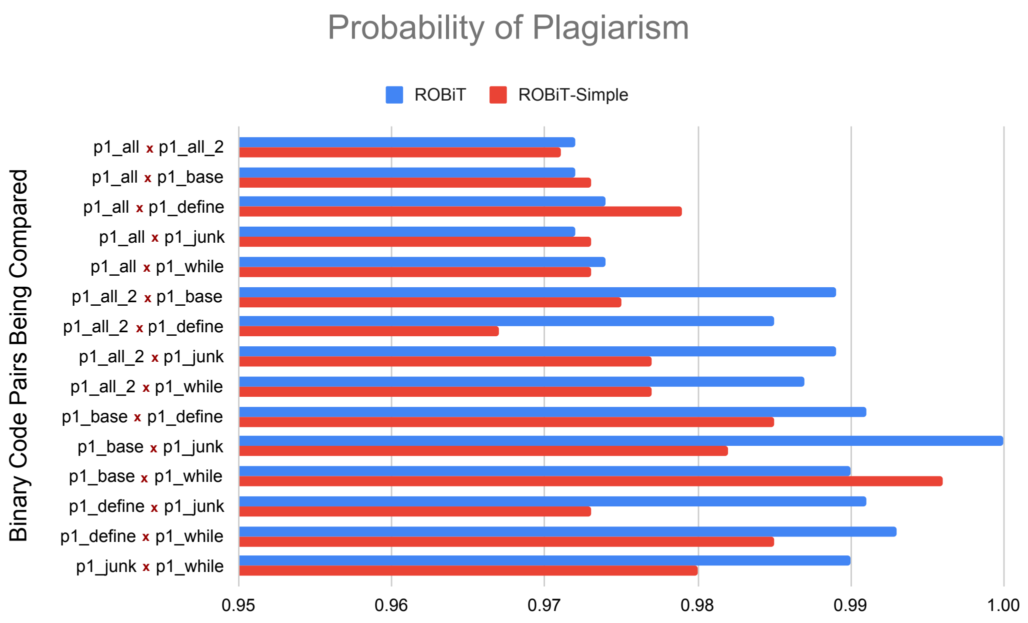


Figura 4. Plagiarism probabilities identified by the techniques `ROBiT` and `ROBiT-Simple`, for each pair of programs analyzed in Experiment 1.

This type of performance pattern, where `ROBiT` outperforms `ROBiT-Simple` only in certain particular situations, but not in all, can be more clearly observed when analyzing the results of Experiment 2 (Figure 5). In this case, it is evident that the method that performs better (when comparing the similarity between a given pair of programs) varies slightly. However, in general, there is a balance of cases in which one of the two methods is slightly better than the other. It is important to note the scale of similarity percentages reported in Figure 5: the scale presented on the x-axis of the figure ranges from 92% similarity to 100% similarity. Any differences in the ability to detect the similarity of each method are small, always varying by less than 10 percentage points.

Considering the observations presented and the possible transformations that can be applied to source codes, it is possible to conclude that the ROBiT method tends to be slightly superior to the ROBiT-Simple method; and that both are significantly more accurate than the state-of-the-art method, MOSS.

4.3. Analysis of the Impact of Syntactic and Semantic Transformations

Observing the performance analysis of MOSS in both experiments, it is notable how applying syntactic and semantic transformations to the original source codes substantially affects its efficacy negatively. Indeed, MOSS has the worst performance among the three methods evaluated in our work, but for a good reason: MOSS is based on token analysis, which makes it particularly susceptible to transformations capable of affecting such characteristics of programs. Our empirical results corroborate this observation, as MOSS proves ineffective in quantifying similarity percentages in 13 out of the 15 cases of transformations presented in the work of [Zhang et al. 2021]—see Figure 1.

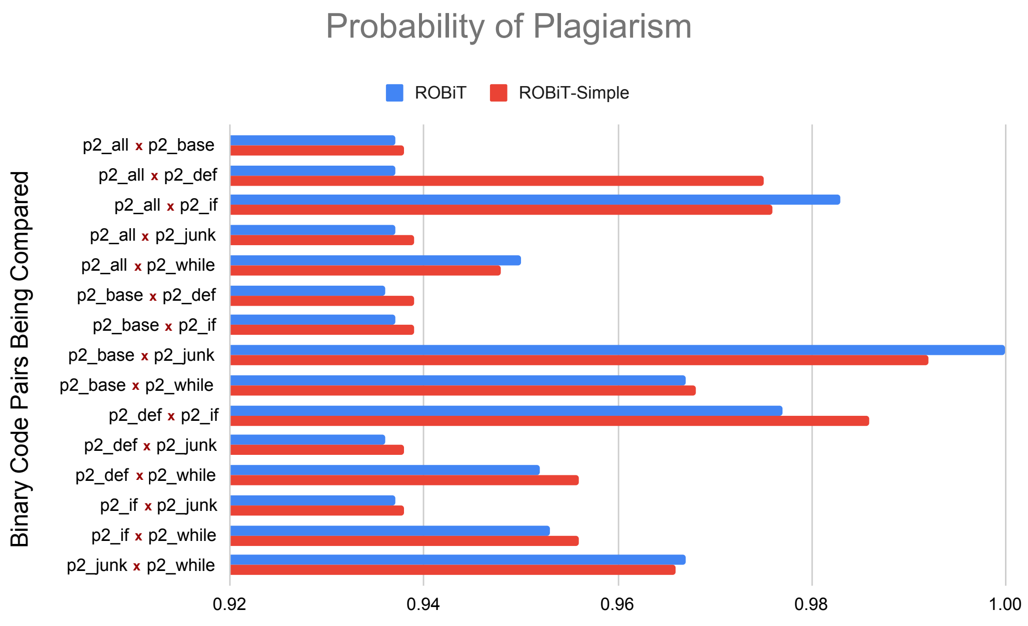


Figura 5. Plagiarism probabilities identified by the techniques ROBiT and ROBiT-Simple, for each pair of programs analyzed in Experiment 2.

Regarding Experiment 1 (Figure 4), we observe that MOSS was able to identify potential plagiarism cases in only two cases: when comparing the files *p1-base* and *p1-define* (52% chance of plagiarism); and the files *p1-base* and *p1-while* (39% chance of plagiarism). Note that these percentages are not visible in Figure 4, as they are significantly lower than the levels of similarity identified by the ROBiT and ROBiT-Simple methods, which are much more effective. The relatively positive performance of MOSS in the two mentioned cases can be understood through manual analysis of the codes in question, resulting from the application of transformations (generated by ChatGPT) of the original source code. In particular, we could observe that in these specific cases, several sections of the codes, although structurally equivalent, remained syntactically similar and, therefore, were (correctly) identified as plagiarisms by MOSS. This suggests that the ChatGPT prompt used in our experiments to generate plagiarized code may not have been detailed enough to enable the automated creation of plagiarism capable of evading similarity detection techniques. This is a point of possible future improvement in our work; in particular, regarding the generation of different types of plagiarism to create test files for evaluating the performance of different anti-plagiarism techniques.

Based on these experiments, we conclude:

- Syntactic and semantic transformations degrade the performance of `MOSS`, often leading to undetected plagiarism.
- `ROBiT` achieves significantly higher detection accuracy and reliability than `MOSS`, particularly under obfuscation.
- `ROBiT`'s performance matches or slightly exceeds that of `ROBiT-Simple` in complex transformation scenarios.
- `ROBiT` is especially robust against **Op13-ChAddJunk** transformations, a common form of semantic plagiarism.

5. Related Work

One of the first syntactic transformations applied by students in the hope of bypassing anti-plagiarism analysis involves changing the order of independent sections of the source code. A series of articles [Wise 1996, Prechelt et al. 2002, Faidhi and Robinson 1987] present syntactic similarity analysis algorithms that emerged in response to this type of simple evasion technique. Generally, they are based on *greedy string tiling* methods, which quantify the similarity between strings and are robust, for example, to the transposition of code sections. Moreover, they are often based on analyzing intermediate code representations in the form of *tokens* and hashing. These methods are effective in the most common cases of plagiarism. Among the techniques that implement this type of anti-plagiarism approach, we emphasize the `MOSS` [Schleimer et al. 2003] tool, widely used in academia and science, and therefore chosen as the starting point of comparison in our experiments (see Section 6.3). In addition to `MOSS`, another syntactic analysis technique for similarity detection was proposed by [Nichols et al. 2019], called JPlag. This technique is based on comparisons using the Smith-Waterman algorithm for sequence alignment, based on analyzing intermediate program representations—particularly their respective abstract syntax trees (ASTs). Unlike our approach, however, which will be built in the context of optimized binary code comparisons, this algorithm identifies potential plagiarism by analyzing higher-level representations derived from the source codes in question.

Several adversarial methods have emerged to evade syntactic analysis. The Mossad framework [Devore-McDonald and Berger 2020], for instance, targets token- and AST-based plagiarism detectors by injecting dead code and other superficial transformations. This type of manipulation interferes with both tokenization and AST construction, rendering syntactic analysis ineffective. While Mossad includes an experimental evaluation involving assembly-level code comparison (compiled with optimization flags), it does not address direct binary-level similarity. In contrast, our method focuses specifically on analyzing compiled executable files rather than intermediate representations or textual outputs like assembly.

Other techniques rely on graph-based representations, which are often more robust than token-based or AST-based methods. For example, [Liu et al. 2006] proposed using program dependency graphs (PDGs) to detect similarity, while [Chae et al. 2013] introduced a valuation-based graph comparison method that assigns numerical similarity scores.

Nevertheless, even these techniques can be undermined by syntactic transformations that subtly alter the relationships among graph nodes.

In the domain of binary similarity and obfuscation, several studies explore how code transformations affect similarity metrics [Luo et al. 2017, DamÁsio et al. 2023, Poornima and Mahalakshmi 2023]. Among them, the work of [Ren et al. 2021] is particularly relevant. Their framework uses aggressive compiler optimization flags, including *-O3*, to intentionally increase the dissimilarity of binary code and evade signature-based detectors. In contrast, our approach inverts this rationale: we apply optimization to reduce syntactic artifacts and normalize functionally equivalent source codes. Additionally, unlike most adversarial frameworks that assume the source code is unavailable, our method operates under the assumption that both original and plagiarized C source files are accessible, as is typical in academic or collaborative environments.

In summary, this section reviewed prior work that is either related to or foundational to this research. Compared to the themes of similarity between source codes using syntactic and semantic analysis algorithms, our method differs by utilizing binary analysis, positioning itself in a scope not explored by typical anti-plagiarism analyzers available on the market. When we look at the binary context itself, our method also differs by not being in the same category of adversarial tests without the presence of the original source code—most works assume a completely independent binary. In addition to using the source code, we also use the opposite idea usually supported for the use of binary optimizations, as instead of using this function as an obfuscation method, in our proposition, it is used as a reversal of syntactic-level obfuscation.

6. Conclusions

Developing effective plagiarism detection techniques is a complex task, given that the affected institutions often lack the necessary resources to deal with the large number of potential infractions. In this work, we investigate methods based on the analysis of binary code similarity and propose a new algorithm, `ROBiT`, which will serve as a complement to existing anti-plagiarism techniques.

As discussed in sections 2, 3, and 5, although several existing methods are effective in various cases, they generally do not explore analysis and comparisons of binary files. This often implies that they are not robust against evasion techniques based on sophisticated semantic and syntactic transformations that would result in graph-based modifications. In this work, we proposed a new anti-plagiarism technique that explores this type of analysis and empirically evaluated various hypotheses related to its efficacy in different scenarios—with varying degrees of sophistication—involving the analysis of adulterated C source codes aimed at circumventing existing plagiarism detection systems.

Through our experimental analysis, we observed the behavior of our method, `ROBiT`, noting its capabilities and limitations. In particular, we were able to experimentally prove that, compared to well-established and widely used methods such as `MOSS`, our method consistently achieved superior results both in terms of the number of plagiarisms discovered and the degree of certainty reported by the technique about the existence of plagiarism.

Moreover, when comparing the efficacy of the `ROBiT` method in relation to a similar technique (`ROBiT-Simple`), which does not explore optimization processes during compilation, we observed similar performance levels. In some test cases, `ROBiT` demonstrates clearly superior performance, but generally, it is not possible to conclude with high statistical reliability that one method is always superior to the other. An exception concerns emblematic cases such as those where plagiarisms are produced via syntactic transformations of the type **Op13-ChAddJunk**. In these cases, our method consistently performed better than both `MOSS` and `ROBiT-Simple`, always being able to recognize plagiarism.

Despite promising results, several limitations merit consideration. First, our current study does not incorporate statistical significance testing. Future work should include formal analysis, such as t-tests or confidence intervals, to better validate the observed performance differences—especially between `ROBiT` and `ROBiT-Simple`. Second, our experiments are based on small-scale, academic-style programs written in C. Evaluating `ROBiT` on diverse code examples, or across multiple programming languages, would help assess its capacity of generalization.

Another key limitation is the use of synthetically generated plagiarized samples via ChatGPT. While this allows controlled experimentation, it may not reflect the full range of strategies employed by real users. To bridge this gap, future studies should incorporate anonymized student submissions or publicly available datasets to evaluate `ROBiT` under real-world conditions.

Finally, we envision potential extensions of this work, including more complex analysis related to intermediate representation structures at the compiler level. We also aim to explore integration with other anti-plagiarism tools to build hybrid detection frameworks that combine binary, syntactic, and structural analysis techniques. Such a combination could significantly enhance detection accuracy, particularly in cases of advanced obfuscation.

In summary, `ROBiT` presents a novel and complementary approach to traditional syntactic plagiarism detection systems by introducing optimization-driven binary analysis into the detection pipeline. While there is room for further development and validation, our results demonstrate the potential of this method to improve detection rates and robustness in the face of increasingly sophisticated plagiarism strategies.

Referências

Aiken, A. (2004). MOSS: A system for detecting software plagiarism. Available at: `https://theory.stanford.edu/~aiken/moss/` (accessed May 2025).

Biderman, S. and Raff, E. (2022). Fooling MOSS detection with pretrained language models. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 2933–2943, New York, NY, USA. Association for Computing Machinery.

Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., and Im, E. G. (2013). Software plagiarism detection: a graph-based approach. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, page 1577–1580, New York, NY, USA. Association for Computing Machinery.

DamÁsio, T., Canesche, M., Pacheco, V., Botacin, M., Faustino da Silva, A., and Quintão Pereira, F. M. (2023). A game-based framework to compare program classifiers and evaders. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, page 108–121, New York, NY, USA. Association for Computing Machinery.

Devore-McDonald, B. and Berger, E. D. (2020). Mossad: defeating software plagiarism detection. *Proc. ACM Program. Lang.*, 4(OOPSLA).

Duracik, M., Hrkut, P., Krsak, E., and Toth, S. (2020). Abstract syntax tree based source code antiplagiarism system for large projects set. *IEEE Access*, 8:175347–175359.

Faidhi, J. and Robinson, S. (1987). An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19.

Free Software Foundation (2024). GCC Command Options: Options That Control Optimization. *GCC Online Documentation*.

Geng, C., Zhang, Y., Pientka, B., and Si, X. (2023). Can chatgpt pass an introductory level functional language programming course? *arXiv preprint arXiv:2305.02230*.

Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics Doklady*, 10:707–710.

Liu, C., Chen, C., Han, J., and Yu, P. S. (2006). GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’06, page 872–881, New York, NY, USA. Association for Computing Machinery.

Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. (2017). Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43(12):1157–1177.

Nichols, L., Dewey, K., Emre, M., Chen, S., and Hardekopf, B. (2019). Syntax-based improvements to plagiarism detectors and their evaluations. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE ’19, page 555–561, New York, NY, USA. Association for Computing Machinery.

Poornima, S. and Mahalakshmi, R. (2023). An inclusive report on robust malware detection and analysis for cross-version binary code optimizations. *International Journal on Recent and Innovation Trends in Computing and Communication*, 11(9):927–937.

Prechelt, L., Malpohl, G., and Philippsen, M. (2002). Finding plagiarisms among a set of programs with jplag. *JUCS - Journal of Universal Computer Science*, 8(11):1016–1038.

Radare2 Team (2024). Binary Diffing (online manual). *Radare2: Libre Reversing Framework for Unix Geeks [GitHub Repository]*.

Ren, X., Ho, M., Ming, J., Lei, Y., and Li, L. (2021). Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 142–157, New York, NY, USA. Association for Computing Machinery.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 76–85, New York, NY, USA. Association for Computing Machinery.

Wise, M. J. (1996). YAP3: Improved Detection of Similarities in Computer Program and Other Texts. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '96, page 130–134, New York, NY, USA. Association for Computing Machinery.

Yenduri, G., Ramalingam, M., ChemmalarSelvi, G., Supriya, Y., Srivastava, G., Maddikunta, P. K. R., DeeptiRaj, G., Jhaveri, R. H., Prabadevi, B., Wang, W., Vasilakos, A. V., and Gadekallu, T. R. (2023). Generative Pre-Trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. *ArXiv preprint ArXiv:2305.10435v2*.

Zhang, W., Guo, S., Zhang, H., Sui, Y., Xue, Y., and Xu, Y. (2021). Challenging machine learning-based clone detectors via semantic-preserving code transformations. *IEEE Transactions on Software Engineering*, 49:3052–3070.